INSTITUTO TECNOLÓGICO Y DE

ESTUDIOS SUPERIORES DE MONTERREY

ACTIVIDAD INTEGRADORA 2:

“EMPRESA A INCURSIONAR EN SERVICIOS DE INTERNET”

Unidad Formativa:

Análisis y diseño de algoritmos avanzados (TC 2038.603)

Profesor:

Adán Octavio Ruíz Martínez

Alumnos:

León Emiliano García Pérez A00573074

Carla Morales López A01639225

Óscar Jahir Valdés Caballero A01638923

Campus:

Guadalajara

Fecha de entrega: Domingo 27 de Noviembre de 2022

“EMPRESA A INCURSIONAR EN SERVICIOS DE INTERNET”

Introducción:

En el presente documento se da documentación al desarrollo de la Actividad Integradora que gira en torno a la implementación de los algoritmos solución para el planteamiento de la Evidencia, se realiza la documentación general propia del desarrollo de programas dentro del Instituto Tecnológico y de Estudios Superiores de Monterrey, así como los casos de prueba en torno a la ejecución del código.

Subcompetencias a Evaluar:

La evidencia es la demostración de lo que se ha logrado. Con ésta, el profesor podrá observar y evaluar la subcompetencia en la profundidad y complejidad (nivel del dominio A, B o C) que la unidad de formación llevó a desarrollar. Con esta evidencia se demuestran las siguientes subcompetencias:

- SICT0101 - Explica el funcionamiento de sistemas computacionales por medio de argumentaciones sustentadas en las interacciones entre los componentes y su entorno creando modelos conceptuales donde se describen los componentes y la relación con su entorno.
- SICT0401 - Aplica los estándares y normas propios de su profesión contrastándolos contra las restricciones de uso de acuerdo al proceso, producto o servicio donde se va a aplicar usando las normas y estándares más relevantes al dominio del problema que se va a resolver, distinguiendo claramente entre ambos.
- STC0101 - Implementa algoritmos computacionales confiables y correctos que solucionan problemas.
- STC0102 - Optimiza algoritmos computacionales robustos y eficientes que se aplican en el desarrollo de soluciones.

Descripción de la Evidencia:

En equipos de máximo 3 personas, se escribe en C++ un programa que ayude a una empresa que quiere incursionar en los servicios de Internet respondiendo a la situación problema 2.

El programa debe:

1. Leer un archivo de entrada que contiene la información de un grafo representado en forma de una matriz de adyacencias con grafos ponderados El peso de cada arista es la distancia en kilómetros entre colonia y colonia, por donde es factible meter cableado. El programa debe desplegar cuál es la forma óptima de cablear con fibra óptica conectando colonias de tal forma que se pueda compartir información entre cualesquiera dos colonias.
2. Debido a que las ciudades apenas están entrando al mundo tecnológico, se requiere que alguien visite cada colonia para ir a dejar estados de cuenta físicos, publicidad, avisos y notificaciones impresos. por eso se quiere saber ¿cuál es la ruta más corta posible que visita cada colonia exactamente una vez y al finalizar regresa a la colonia origen? El programa debe desplegar la

ruta a considerar, tomando en cuenta que la primera ciudad se le llamará A, a la segunda B, y así sucesivamente

3. El programa también debe leer otra matriz cuadrada de N x N datos que representan la capacidad máxima de transmisión de datos entre la colonia i y la colonia j. Como estamos trabajando con ciudades con una gran cantidad de campos electromagnéticos, que pueden generar interferencia, ya se hicieron estimaciones que están reflejadas en esta matriz. La empresa quiere conocer el flujo máximo de información del nodo inicial al nodo final. Esto debe desplegarse también en la salida estándar.
4. Teniendo en cuenta la ubicación geográfica de varias "centrales" a las que se pueden conectar nuevas casas, la empresa quiere contar con una forma de decidir, dada una nueva contratación del servicio, cuál es la central más cercana geográficamente a esa nueva contratación. No necesariamente hay una central por cada colonia. Se pueden tener colonias sin central, y colonias con más de una central.

Entrada:

- Un numero entero N que representa el número de colonias en la ciudad.
- Matriz cuadrada de N x N que representa el grafo con las distancias en kilómetros entre las colonias de la ciudad.
- Matriz cuadrada de N x N que representa las capacidades máximas de flujo de datos entre colonia i y colonia j.
- Lista de N pares ordenados de la forma (A,B) que representan la ubicación en un plano coordenado de las centrales.

Salida:

Forma de cablear las colonias con fibra (lista de arcos de la forma (A,B)).

Ruta a seguir por el personal que reparte correspondencia, considerando inicio y fin en la misma colonia.

Valor de flujo máximo de información del nodo inicial al nodo final.

Lista de polígonos (cada elemento es una lista de puntos de la forma (x,y)).

```
Ejemplo de Input:

4



0 16 45 32

16  0 18 21

45 18  0  7

32 21  7  0
```

```
0 48  12  18

52  0 42 32

18 46  0 56

24 36 52  0
```

```
(200,500)

(300,100)

(450,150)

(520,480)
```

Especificaciones:

Entrega en este espacio en canvas un archivo .ZIP llamado A0XXXXXXX_ActInt2, (donde las XXXXXXXs son la matrícula de alguno de los integrantes del equipo) que contenga dentro una carpeta llamada A0XXXXXXX_ActInt2, en donde se encontrará UN único archivo .cpp. Se pueden tener uno o más archivos .h.

Incluye un archivo team.txt que incluya los nombres de todos los integrantes del equipo, con sus matrículas y las partes específicas del proyecto en las que trabajó cada quien

Incluye también los archivos ReflexActInt2_A0XXXXXXX.pdf

Ten en cuenta que tu programa se probará con casos de prueba grandes.

Se tendrá una sesión de revisión donde a cada integrante del equipo se le harán preguntas específicas y todos deben saber cómo funciona cada parte de la solución propuesta.

Especificaciones:

Para la evaluación de las evidencias se hará uso de rúbricas generales con descriptores que muestran desempeños clave y criterios que distinguen tu grado de logro en las subcompetencias, ya sea Destacado, Sólido, Básico o Incipiente.

La evidencia tiene impacto en tu calificación, para ello se tomará en cuenta:

Para obtener el **100%** de los puntos de esta actividad, se divide en:

- **80%** - Cumple correcta y eficientemente con la funcionalidad requerida por parte de la actividad:
    - **20%** - primera parte del output (forma de cableado óptimo).
    - **20%** - segunda parte del output (ruta para repartir correspondencia)
    - **20%** - tercera parte del output (substring más largo común)
    - **20%** - cuarta parte del output (lista de polígonos)
- **15%** - El documento de reflexión incluye la explicación de diferentes algoritmos aplicados a esta situación problema, así como la complejidad computacional de cada uno de ellas.
- **05%** - El código deberá seguir los lineamientos estipulados en el estándar de codificación.

Sección de la Calidad de Software:

Identación:

Se realiza la debida Identación (Sangrado) adecuadamente, en torno al Manual Estándar de Programación del Instituto Tecnológico y de Estudios Superiores de Monterrey.

Documentación:

Se realiza la documentación adecuadamente, en torno al Manual Estándar de Programación del Instituto Tecnológico y de Estudios Superiores de Monterrey. Con respecto a los comentarios propios de la documentación del Programa se encuentran los siguientes Datos Generales:

- Programa que: Implementa los algoritmos solución para la Situación Problema.
- Programadores: León Emiliano García Pérez [A00573074], Carla Morales López [A01639225], Óscar Jahir Valdés Caballero [A01638923].
- Fecha de entrega: Domingo 27 de Noviembre de 2022.

Mantenibilidad:

En cuestión de Mantenibilidad el código al emplear el paradigma funcional, permite que el código sea más fácil de corregir o reparar en presencia de errores, así como al probar los casos de prueba; ya que al corregir la función que poseé el error, se logra corregir en todas las demás funciones en que ésta está presente. Tal y como se pudo comprobar durante el desarrollo del presente código.

Funcionamiento en el Código:

Inclusiones de librerías:

- iostream
- vector
- algorithm
- queue
- stack

Ajuste:

- using namespace std;

Funciones generales:

- void espacio():

Estructuras:

- Punto
  - int x;
  - int y;

Definición:

- Edge
  - pair
    - int
    - int

Definición de variables globales:

- Punto punto0;

Facilidad de Lectura y Rendimiento:

Funciones:

- vector <pair<int,edge>> edges(vector<vector<int>> matrix): Constructor desde una matriz de adyacencia.
- int findSet(int i, vector<int> parent): Función que encuentra el set de un vertice.
- vector<pair<int,edge>> kruskal (vector<pair<int,edge>> edges, int numV): Función que implementa el algoritmo de Kruskal para la búsqueda del MST (Minimum Spanning Tree).
- void printMST(vector<pair<int,edge>> mst, int numV): Función que imprime el Output del MST.
- pair <int,string> pathCost (vector<int> set, int end, vector<vector<int>> cost, vector<pair<int,string>> & bitmasks): Función que retorna el mínimo costo y la ruta de un path que visita todos los nodos exactamente una vez, iniciando en 0 y terminando en el nodo dado.
- pair<int,string> tsp (vector<vector <int>> matrix): Función que retorna el costo mínimo y la ruta de un ciclo que visita todos los nodos dados exactamente una vez.
- void espacio(): Función que imprime en consola un salto de línea, no recibe parámetros y no tiene valor de retorno.
- bool bfs(vector<vector<int>>& rMatriz, int origen, int destino, vector<int>& parentezco): Implementación de Búsqueda en Anchura, recibe como parámetro un vector de vectores de enteros referenciado, que funge como matríz, un entero para el nodo origen, un entero para el nodo destino, y un vector de enteros referenciado con el parentezco.
- void fordFulkerson(vector<vector<int>>& matriz, int origen, int destino): Implementación del Algoritmo Ford-Fulkerson, recibe como parámetro un

vector de vectores de enteros referenciado, que funge como matríz, un entero para el nodo origen y un entero para el nodo destino.
- Punto siguientoAlTop(stack<Punto>& pila): Función que ayuda a encontrar el punto siguiente al top de un stack, recibe un stack referenciado de puntos y retorno el Punto siguiente al Top.
- void intercambio(Punto& punto1, Punto& punto2): Función que intercambia dos puntos, recibe la referencia a dos puntos, no tiene valor de retorno.
- int distanciaCuadrada(Punto punto1, Punto punto2): Función que calcula el cuadrado de la distancia entre dos puntos, recibe como parámetro los dos puntos, y retorna el entero de la distancia cuadrada.
- int direccion(Punto p, Punto q, Punto r): Función que determina el sentido dados tres puntos, recibe los tres puntos y retorna un 0 si es Collinear, 1 si es Dextrógiro (Al sentido del Reloj), 2 si es Levógiro (Al sentido ContraReloj).
- int comparar(const void* voidPunto1, const void* voidPunto2): Función auxiliar de comparación para ordenar un arreglo de puntos respecto al primero, recibe dos apuntadores a constantes void y retorna -1 o 1 según sea el caso.
- void cascaraConvexaGraham(vector<Punto>& puntos, int n): Función que imprime la cáscara conveza dado un arreglo de puntos, recibe además la cantidad n de puntos, no tiene valor de retorno.
- int main(): Función main de ejecución de código, que implementa los algoritmos solución para la Situación Problema de la Actividad Integradora 2.

En cuanto a la Facilidad de Lectura es sencillo identificar qué es lo que realiza cada función debido al propio nombre del método, y en cuanto a parámetros también es identificable conocer qué parámetro es el esperado.

En cuanto al Rendimiento, se toma en cuenta la Complejidad Computacional de las funciones, cada determinación de complejidad está fundamentada en el análisis del comportamiento de cada función con especial énfasis en los ciclos que presentan cada una de éstas:

<u>Complejidad Computacional:</u>

- vector <pair<int,edge>> edges(vector<vector<int>> matrix): O(v^2), v = número de vértices.
- int findSet(int i, vector<int> parent): O(1).
- vector<pair<int,edge>> kruskal (vector<pair<int,edge>> edges, int numV): O(E log E), E = número de aristas.
- void printMST(vector<pair<int,edge>> mst, int numV): O(v), v = número de vértices.
- pair <int,string> pathCost (vector<int> set, int end, vector<vector<int>> cost, vector<pair<int,string>> & bitmasks): O(n * 2^n), n = número de nodos.
- pair<int,string> tsp (vector<vector <int>> matrix): O(n^2 * 2^n), n = número de nodos.
- void espacio(): O(1).
- bool bfs(vector<vector<int>>& rMatriz, int origen, int destino, vector<int>& parentezco):O(VE^2), V = cantidad de Vértices y E = cantidad de Aristas.
- void fordFulkerson(vector<vector<int>>& matriz, int origen, int destino): O(EV^3), V = cantidad de Vértices y E = cantidad de Aristas.
- Punto siguientoAlTop(stack<Punto>& pila): O(1).

- void intercambio(Punto& punto1, Punto& punto2): O(1).
- int distanciaCuadrada(Punto punto1, Punto punto2): O(1).
- int direccion(Punto p, Punto q, Punto r): O(1).
- int comparar(const void* voidPunto1, const void* voidPunto2): O(1).
- void cascaraConvexaGraham(vector<Punto>& puntos, int n): Complejidad Computacional: O(n*log(n)), n = cantidad de puntos.
- int main(): Complejidad Máxima de los Algoritmos Implementados.

Complejidad Computacional [Complejidad Mayor, fundamentados en la Jerarquía de Complejidad]:

$$1 << \log n << n << n \log n << n^2 << n^3 << \ldots\ldots << 2^n << n!$$

Casos de Prueba:

CASO DE PRUEBA CERO: PROPUESTO POR LA ACTIVIDAD

ARCHIVO DE TEXTO: in.txt

```
4


0 16 45 32
16  0 18 21
45 18  0  7
32 21  7  0


0 48  12  18
52  0 42 32
18 46  0 56
24 36 52  0


(200,500)
(300,100)
(450,150)
(520,480)
```

OUTPUT GENERADO:

```
----- INPUT: -----


----- OUTPUT: -----

--- Algoritmo de Kruskal ---


La forma mas optima de conectar todas las colonias con fibra optima en una sola red es: (C, D), (A, B), (B, C)

El total de kilometros de fibra optica requerida es: 41




--- TSP (Dynamic Programming) ---


El recorrido mas corto que visita cada colonia y vuelve al origen es: A -> D -> C -> B -> A

La distancia total del recorrido en kilometros es: 73




--- Algoritmo de Ford-Fulkerson ---


El flujo maximo de información transmisble entre el primer y el último nodo es: 78




--- Convex Hull - Escaneo de Graham ---


Las centrales de la ciudad se encuentran en los puntos:
```

```
(300,100)

(450,150)

(520,480)

(200,500)
```

Sobre el presente entregable y su contenido.

Dentro del ZIP en el cual viene adjunto el presente documento se encontrará los siguientes archivos:

1. a.out : Ejecutable.
2. ActInt2-A00573074-A01639225-A01638923.docx: Documento de Word con el presente documento.
3. ActInt2-A00573074-A01639225-A01638923.pdf: Documento PDF con el presente documento.
4. in.txt: Archivo de Texto para casos de prueba.
5. main.cpp : Desarrollo en código de la actividad.
6. ReflexActInt2-A00573074.pdf: Documento PDF con la Reflexión del Alumno.
7. ReflexActInt2-A01639225.pdf: Documento PDF con la Reflexión del Alumno.
8. ReflexActInt2-A01638923.pdf: Documento PDF con la Reflexión del Alumno.
9. team.txt : Matrícula, nombre y desarrollo de cada miembro del equipo.

Referencias consultadas:

Para el desarrollo del código, se recurrió principalmente a la consulta de la siguiente fuente:

- “Kruskal's Algorithm”, online, Programiz, online, Programiz, available on https://www.programiz.com/dsa/kruskal-algorithm (26.11.2022).
- (2022), “Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming)”, online, Geeks For Geeks, available on https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/ (26.11.2022).
- (2022), “Ford-Fulkerson Algorithm for Maximum Flow Problem”, online, Geeks For Geeks, available on https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/ (26.11.2022).
- (2022), “Convex Hull | Set 2 (Graham Scan)”, online, Geeks For Geeks, available on https://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/ (26.11.2022).

Desarrollo main.cpp:

```
// Programa que: Implementa los algoritmos solución para la Situación Problema de la Actividad Integradora 2.

// Programadores: León Emiliano García Pérez [A00573074], Carla Morales López [A01639225], Óscar Jahir Valdés Caballero [A01638923].

// Fecha de entrega: Domingo 27 de Noviembre de 2022.
```

```
//    ------------------------------------    INCLUSIÓN    DE    LIBRERÍAS
--------------------------------------

#include <iostream>

#include <vector>

#include <algorithm>

#include <queue>

#include <stack>



//              ----------------------------------              KRUSKAL
----------------------------------------



#define edge std::pair<int, int>



// Constructor from an adjacency matrix.

// Time complexity: O(v^2) for V: number of vertices.

std::vector<std::pair<int,  edge>  >  edges(std::vector<  std::vector<int>
> matrix)  {

    std::vector<std::pair<int, edge> > edges;

    int numV = matrix.size();

    for (int i = 0; i < numV; i++) {

        for (int j = 0; j < numV; j++) {

            if (matrix[i][j] != 0) {

                    edges.push_back(std::make_pair(matrix[i][j], edge(i,
j)));

            }

        }

    }

    return edges;

}



// Finds the set of a vertex.
```

```
// Time complexity: O(1)

int findSet(int i, std::vector<int> parent) {

    // Recursive function to find the parent of a node.



    // Base case: if the node is its own parent, return it.

    if (i == parent[i]) {

        return i;

    } else {

     // Recursive case: if the node is not its own parent, find the
parent of the parent.

        return findSet(parent[i], parent);

    }

}



// Kruskal's algorithm for finding the minimum spanning tree (MST).

// Time complexity: O(E log E) for E: number of edges.

std::vector<std::pair<int,  edge>  >  kruskal(std::vector<std::pair<int,
edge> > edges, int numV) {

    std::vector<int> parent;

    parent.resize(numV);

    for (int i = 0; i < numV; i++) {

        parent[i] = i;

    }

    std::vector<std::pair<int, edge> > mst;

    int i, setV1, setV2;

     std::sort(edges.begin(), edges.end());  // Sort edges by weight in
ascending order.

     for (i = 0; i < edges.size(); i++) { // Iterate through all the
edges.

        // edges[i][0] = weight

        // edges[i][1] = edge
```

```
        setV1 = findSet(edges[i].second.first, parent);

        setV2 = findSet(edges[i].second.second, parent);



         // If there are no cycles (the nodes are not in the same set),
add the edge to the MST.

        if (setV1 != setV2) {

            mst.push_back(edges[i]);  // Add to tree.

            parent[setV1] = parent[setV2]; // Link the nodes' sets.

        }

    }

    return mst;

}



// This function prints the output of the MST.

// Time complexity: O(V) for V: number of vertices.

void printMST( std::vector<std::pair<int, edge> > mst, int numV) {

    int size = mst.size();

    for (int i = 0; i < size - 1; i++)

    {

          std::cout << "(" << char(mst[i].second.first + 65) << ", " <<
char(mst[i].second.second + 65) << "), ";

    }

     std::cout << "(" << char(mst[size - 1].second.first + 65) << ", "
<< char(mst[size - 1].second.second + 65) << ")" << std::endl;



    // Print the cost.

    int cost = 0;

    for (int i = 0; i < mst.size(); i++) {

        cost += mst[i].first;

    }
```

```
    std::cout << "El total de kilometros de fibra optica requerida es:
" << cost << std::endl;

}




//                ----------------------------------                TSP
--------------------------------------



// This function returns the minimum cost (and the route) of a path
that visits all given nodes exactly once, starting at 0, and ending at
the given node.

// Time complexity: O(n * 2^n) for n: number of nodes.

std::pair<int,  std::string>  pathCost(std::vector<int>  set,  int  end,
std::vector<  std::vector<int>  >  costs,  std::vector<std::pair<int,
std::string>> &bitmasks) {

    int size = set.size();

    std::string path;

    char endC(end + 65);



    // If the set has only two elements, return the distance form 0 to
the end element.

    if (size == 2) {

        path = "A -> ";

        path += endC;

        path  += " -> ";

        return std::make_pair(costs[0][end], path);

    }

     // Otherwise, find the minimum cost of the set without the end
element and every element of the set plus the distance from the element
to the first node.

    int minCost = 0;



    int mask = 0;

    // Remove the end element from the set.
```

```
    std::vector<int> newSet;

    for (int i = 0; i < size; i++) {

        if (set[i] != end) {

            newSet.push_back(set[i]);

            mask += 1 << set[i];

        }

    }



     // Search the cost in the Dynamic Programming table. Return it if
it exists.

    if (bitmasks[mask].first != 0) {

        return bitmasks[mask];

    }



    // If the result has not been calculated, calculate it.

    for (int i = 1; i < newSet.size(); i++) {

                std::pair<int,  std::string>  subPath  =  pathCost(newSet,
newSet[i], costs, bitmasks);

        int cost = subPath.first + costs[newSet[i]][end];

        if (minCost == 0 || cost < minCost) {

            minCost = cost;

            path = subPath.second;

            path += endC;

            path += " -> ";

        }

    }



    // Save the result in the Dynamic Programming table.

    bitmasks[mask] = std::make_pair(minCost, path);
```

```
    return std::make_pair(minCost, path);

}



// This function returns the minimum cost (and the route) of a cycle
that visits all given nodes exactly once.

// Time complexity: O(n^2 * 2^n) for n: number of nodes.

std::pair<int, std::string> tsp(std::vector< std::vector<int> > matrix)
{

    std::vector<std::pair<int, std::string> > cycleCosts;

    std::vector<int> set;

    for (int i = 0; i < matrix.size(); i++) {

        set.push_back(i);

    }

    // Generate bitmask vector.

          std::vector<std::pair<int,  std::string>  >  bitmasks(1  <<
matrix.size(), std::make_pair(0, ""));



    for (int i = 1; i < matrix.size(); i++) {

          // Find minimum cost path starting from vertex 1, passing
through every node once and ending at vertex i.

        // The cost of the Hamiltonian cycle is the sum of the minimum
cost path and the cost of the edge from the last node to the first
node.

         std::pair<int, std::string> subPath = pathCost(set, i, matrix,
bitmasks);

        int totalCost = subPath.first + matrix[i][0];

         cycleCosts.push_back(std::make_pair(totalCost, subPath.second +
"A"));

    }

    // Find the minimum cost Hamiltonian cycle.

     sort(cycleCosts.begin(), cycleCosts.end(), [](const std::pair<int,
std::string>& a, const std::pair<int, std::string>& b) {

        return a.first < b.first;

    });
```

```
    return cycleCosts[0];

}



// ------------------------------------------------- AJUSTE A ESTANDAR
-------------------------------------------------



//Ajuste a estandar.

using namespace std;



//     ------------------------------------------------     FUNCIONES
GENERALES ------------------------------------------------



// Función que imprime en consola un salto de línea, no recibe
parámetros y no tiene valor de retorno.

// Complejidad Computacional: O(1)

void espacio() {

    cout << endl;

}



//   ------------------------------------------------   FORD-FULKERSON
ALGORITHM ------------------------------------------------



//Implementación de Búsqueda en Anchura, recibe como parámetro un
vector de vectores de enteros referenciado, que funge como matríz, un
entero para el nodo origen, un entero para el nodo destino, y un vector
de enteros referenciado con el parentezco.

//Complejidad Computacional: O(VE^2) Siendo V la cantidad de Vértices y
E la cantidad de Aristas.

bool  bfs(vector<vector<int>>&  rMatriz,  int  origen,  int  destino,
vector<int>& parentezco){



    int vertices;

    queue<int> cola;
```

```
    vertices = rMatriz.size();


    vector<bool> visitados(vertices, false);


    cola.push(origen);
    visitados[origen] = true;
    parentezco[origen] = -1;


    while (!cola.empty()){
        int u;
        u = cola.front();
        cola.pop();


        for (int v = 0; v < vertices; v++){
            if (visitados[v] == false && rMatriz[u][v] > 0){
                if (v == destino){
                    parentezco[v] = u;
                    return true;
                }
                cola.push(v);
                parentezco[v] = u;
                visitados[v] = true;
            }
        }


    }


    return false;
}
```

```
// Implementación del Algoritmo Ford-Fulkerson,  recibe como parámetro un vector de vectores de enteros referenciado, que funge como matríz, un entero para el nodo origen y un entero para el nodo destino.

// Complejidad Computacional: O(EV^3) Siendo V la cantidad de Vértices y E la cantidad de Aristas.

void fordFulkerson(vector<vector<int>>& matriz, int origen, int destino){



    int u;

    int v;

    int vertices;

    int flujoMaximo;



    vertices = matriz.size();



    vector<vector<int>> rMatriz(vertices,vector<int>(vertices));

    vector<int> parentezco(vertices);



    for (u = 0; u < vertices; u++){

        for (v = 0; v < vertices; v++){

            rMatriz[u][v] = matriz[u][v];

        }

    }



    flujoMaximo = 0;



      // Complejidad Computacional: O(VE^2) Siendo V la cantidad de Vértices y E la cantidad de Aristas.

    while(bfs(rMatriz,0,vertices-1,parentezco)){



        int trayectoriaDeFlujo;
```

```
        trayectoriaDeFlujo = INT32_MAX;


        for (v = destino; v!= origen; v = parentezco[v]){
            u = parentezco[v];
            trayectoriaDeFlujo = min(trayectoriaDeFlujo,rMatriz[u][v]);
        }


        for (v = destino; v != origen; v = parentezco[v]){
            u = parentezco[v];
            rMatriz[u][v] -= trayectoriaDeFlujo;
            rMatriz[v][u] += trayectoriaDeFlujo;
        }


        flujoMaximo += trayectoriaDeFlujo;


    }


    espacio();
     cout << "El flujo maximo de información transmisble entre el primer y el último nodo es: " << flujoMaximo << endl;
    espacio();


}


//  ------------------------------------------------  CONVEX  HULL  - GRAHAM SCAN ------------------------------------------------


//Definición de estrucutra Punto. Poseé un entero x y un entero y.
struct Punto {
    int x;
```

```
    int y;

};



//Punto global que funciona como auxiliar para realizar un ordenamiento.

Punto punto0;



// Función que ayuda a encontrar el punto siguiente al top de un stack, recibe un stack referenciado de puntos y retorno el Punto siguiente al Top.

// Complejidad Computacional: O(1)

Punto siguientoAlTop(stack<Punto>& pila) {



    Punto punto;

    Punto resultado;



    punto = pila.top();

    pila.pop();

    resultado = pila.top();

    pila.push(punto);



    return resultado;

}



// Función que intercambia dos puntos, recibe la referencia a dos puntos, no tiene valor de retorno.

//Complejidad Computacional: O(1)

void intercambio(Punto& punto1, Punto& punto2) {



    Punto auxiliar;
```

```
    auxiliar = punto1;

    punto1 = punto2;

    punto2 = auxiliar;

}



// Función que calcula el cuadrado de la distancia entre dos puntos, recibe como parámetro los dos puntos, y retorna el entero de la distancia cuadrada.

//Complejidad Computacional: O(1)

int distanciaCuadrada(Punto punto1, Punto punto2) {

    return (punto1.x - punto2.x) * (punto1.x - punto2.x) + (punto1.y - punto2.y) * (punto1.y - punto2.y);

}



// Función que determina el sentido dados tres puntos, recibe los tres puntos y retorna un 0 si es Collinear, 1 si es Dextrógiro (Al sentido del Reloj), 2 si es Levógiro ( Al sentido ContraReloj).

// Complejidad Computacional: O(1)

int direccion(Punto p, Punto q, Punto r) {



    int valor;



    valor = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);



    if (valor == 0) {

        return 0;

    }

    else if (valor > 0) {

        return 1;

    }

    else {

        return 2;
```

```
    }

}



// Función auxiliar de comparación para ordenar un arreglo de puntos respecto al primero, recibe dos apuntadores a constantes void y retorna -1 o 1 según sea el caso.

// Complejidad Computacional: O(1)

int comparar(const void* voidPunto1, const void* voidPunto2) {



    int sentido;

    Punto* punto1;

    Punto* punto2;



    punto1 = (Punto*)voidPunto1;

    punto2 = (Punto*)voidPunto2;

    sentido = direccion(punto0, *punto1, *punto2);



    if (sentido == 0) {

                        if   (distanciaCuadrada(punto0,   *punto2)   >= distanciaCuadrada(punto0, *punto1)) {

            return -1;

        }

        else {

            return 1;

        }

    }

    else if (sentido == 2) {

        return -1;

    }

    else {

        return 1;
```

```
    }

}



// Función que imprime la cáscara conveza dado un arreglo de puntos, recibe además la cantidad n de puntos, no tiene valor de retorno.

// Complejidad Computacional: O(n*log(n)), siendo n la cantidad de puntos.

void cascaraConvexaGraham(vector<Punto>& puntos, int n) {



    int yMinima;

    int minimo;

    int tamanio;

    stack<Punto> pilaResultante;

    vector<Punto> resultado;



    yMinima = puntos[0].y;

    minimo = 0;



    // Complejidad Computacional: O(n), siendo n la cantidad de puntos.

    for (int i = 1; i < n; i++) {



        int y;

        y = puntos[i].y;



            if ((y < yMinima) || (yMinima == y && puntos[i].x < puntos[minimo].x)) {

            yMinima = puntos[i].y;

            minimo = i;

        }

    }
```

```
    intercambio(puntos[0], puntos[minimo]);

    punto0 = puntos[0];



     // Complejidad Computacional: O(n*log(n)), siendo n la cantidad de
puntos.

    qsort(&puntos[1], n - 1, sizeof(Punto), comparar);



    tamanio = 1;



    // Complejidad Computacional: O(n), siendo n la cantidad de puntos.

    for (int i = 1; i < n; i++) {



         while (i < n - 1 && direccion(punto0, puntos[i], puntos[i + 1])
== 0) {

             i++;

        }



        puntos[tamanio] = puntos[i];

        tamanio++;

    }



    if (tamanio < 3) {

        espacio();

        cout << "¡¡¡Imposible la creacion de un poligono convexo!!!";

        espacio();

        espacio();

        return;

    }



    pilaResultante.push(puntos[0]);
```

```
    pilaResultante.push(puntos[1]);

    pilaResultante.push(puntos[2]);



     //Complejidad Computacional: O(m), siendo m el tamaño del arreglo
modificado.

    for (int i = 3; i < tamanio; i++) {



                            while    (pilaResultante.size()    >    1    &&
direccion(siguientoAlTop(pilaResultante),          pilaResultante.top(),
puntos[i]) != 2) {

            pilaResultante.pop();

        }



        pilaResultante.push(puntos[i]);

    }



    while (!pilaResultante.empty()) {

        Punto p = pilaResultante.top();

        resultado.push_back(p);

        pilaResultante.pop();

    }



    espacio();



     //Los resultados se brindan en orden inverso a las manecillas del
reloj.

    //cout << "POLIGONO CONVEXO MAS PEQUENIO: [PUNTOS DADOS EN SENTIDO
CONTRARIO A LAS MANECILLAS DEL RELOJ]" << endl;



     // Complejidad Computacional: O(v), siendo v el tamaño del vector
resultado.

    for (int i = resultado.size() - 1; i >= 0; i--) {
```

```
        cout << "(" << resultado[i].x << "," << resultado[i].y << ") ";

        espacio();

    }



    espacio();

}



// ------------------------------------------------- MAIN: DRIVER CODE
-------------------------------------------------



//Función main de ejecución de código, que implementa los algoritmos
solución para la Situación Problema de la Actividad Integradora 2.

int main() {



    int n; //Cantidad de Colonias en la Ciudad

    vector<vector<int>> distancias; //Distancias entre colonias.

    vector<vector<int>> flujos; //Capacidad Máxima de Flujos.

     vector<Punto> centrales; //Ubicación en un plano coordenado de las
centrales.



    cout << "----- INPUT: -----" << endl;

    n = 0;

    cin >> n;



    for (int i = 0; i < n; i++) {

        vector<int> auxiliar;

        for (int j = 0; j < n; j++) {

            int temporal;

            cin >> temporal;

            auxiliar.push_back(temporal);

        }
```

```
        distancias.push_back(auxiliar);
    }


    for (int i = 0; i < n; i++) {
        vector<int> auxiliar;
        for (int j = 0; j < n; j++) {
            int temporal;
            cin >> temporal;
            auxiliar.push_back(temporal);
        }
        flujos.push_back(auxiliar);
    }


    for (int i = 0; i < n; i++) {
        string punto;
        int coma;
        Punto auxiliar;
        int temporalX;
        int temporalY;
        cin >> punto;
        punto = punto.substr(1, punto.size() - 2);
        coma = punto.find(',');
        temporalX = stoi(punto.substr(0, coma));
        temporalY = stoi(punto.substr(coma+1,punto.size()-coma));
        auxiliar.x = temporalX;
        auxiliar.y = temporalY;
        centrales.push_back(auxiliar);
    }
```

```
    espacio();



    cout << "----- OUTPUT: -----" << endl;



    //El programa debe desplegar cuál es la forma óptima de cablear con fibra óptica conectando colonias de tal forma que se pueda compartir información entre cuales quiera dos colonias.

    cout << "--- Algoritmo de Kruskal ---" << endl;

    std::vector<std::pair<int, edge> > graph = edges(distancias);

    std::vector<std::pair<int, edge> > mst = kruskal(graph, n);

     std::cout << std::endl << "La forma mas optima de conectar todas las colonias con fibra optima en una sola red es: ";

    printMST(mst, n);

    std::cout << "\n\n";



      // Debido a que las ciudades apenas están entrando al mundo tecnológico, se requiere que alguien visite cada colonia para ir a dejar estados de cuenta físicos, publicidad, avisos y notificaciones impresos. por eso se quiere saber ¿cuál es la ruta más corta posible que visita cada colonia exactamente una vez y al finalizar regresa a la colonia origen?

    cout << "--- TSP (Dynamic Programming) ---" << endl;

    std::pair<int, std::string> minHamilton = tsp(distancias);

     cout << endl << "El recorrido mas corto que visita cada colonia y vuelve al origen es: " << minHamilton.second << endl;

     cout << "La distancia total del recorrido en kilometros es: " << minHamilton.first << "\n\n\n";



     // La empresa quiere conocer el flujo máximo de información del nodo inicial al nodo final. Esto debe desplegarse también en la salida estándar.

    cout << "--- Algoritmo de Ford-Fulkerson ---" << endl;

    fordFulkerson(flujos,0,n-1);

    std::cout << std::endl;
```

```
    cout << "--- Convex Hull - Escaneo de Graham ---" << endl;

     std::cout << std::endl << "Las centrales de la ciudad se encuentran
en los puntos: " << std::endl;

    cascaraConvexaGraham(centrales,n);

    std::cout << std::endl;



    return 0;

}
```